
# Continuously Measuring Critical Section Pressure with the Free Lunch Profiler

**Florian David, Gaël Thomas, Julia Lawall , Gilles Muller**

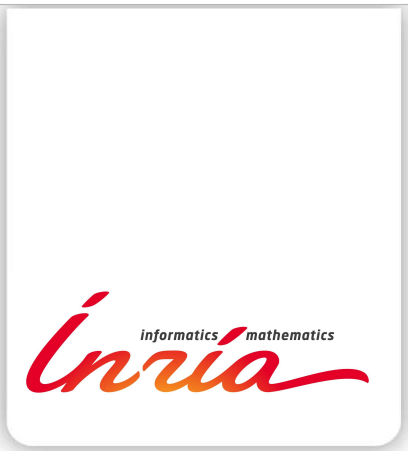
**RESEARCH REPORT**

**N° 8486**

Février 2014

Project-Team REGAL

ISSN 0249-6399 ISRN INRIA/RR--8486--FR+ENG




# Continuously Measuring Critical Section Pressure with the Free Lunch Profiler

Florian David *, Gaël Thomas *, Julia Lawall †, Gilles Muller †

Project-Team REGAL

Research Report n° 8486 — Février 2014 — 24 pages

**Abstract:** Today, Java is regularly used to implement large multi-threaded server-class applications, whose scalability could be hampered by the execution of critical sections. Profiling is needed to identify critical sections that are problematic. However, profiling such applications is challenging, due to their long running times and the range of possible runtime conditions.

We propose Free Lunch, a new profiler designed to identify locks and critical sections that hamper scalability. Free Lunch is designed around a new metric, *critical section pressure*, and can be used in-vivo, while the application is run by end-users.

Using Free Lunch, we have identified a contention phase in the distributed Cassandra NoSQL database and in several applications from the DaCapo benchmark suite. On the latter, we were able to improve the performance of the Xalan benchmark by 15%. In an evaluation on over thirty applications, we found that the overhead of Free Lunch is never greater than 6%.

**Key-words:** Java, Locks, Profiler, Multicore architecture

* LIP6/INRIA
† INRIA/LIP6

**RESEARCH CENTRE**
**PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

# Mesure continuelle de la Critical Section Pressure avec le profileur Free Lunch

**Résumé :** Aujourd'hui, Java est régulièrement utilisé pour implémenter des applications fortement multi-threadées de classe serveur dont le passage à l'échelle pourrait être freiné par l'exécution des sections critiques. Le profilage est nécessaire pour identifier les sections critiques qui posent problème. Cependant, le profilage de telles applications est difficile à cause de leur longue durée d'exécution et de l'étendue des conditions d'exécutions possibles.

Nous présentons Free Lunch, un nouveau profileur conçu pour identifier les verrous et les sections critiques qui freinent le passage à l'échelle. Free Lunch est conçu autour d'une nouvelle métrique appelée *critical section pressure* et peut être utilisé in-vivo pendant que l'application est exécutée par les utilisateurs finaux.

En utilisant Free Lunch, nous avons identifé une phase de contention dans la base de données distribuée NoSQL Cassandra ainsi que dans plusieurs applications provenant de la suite d'applications DaCapo. Dans cette dernière, nous avons été capable d'améliorer les performances de l'application Xalan de 15%. Parmi une évaluation de plus de 30 applications, nous avons trouvé que le surcoût d'exécution de Free Lunch n'est jamais plus important que 6%.

**Mots-clés :** Java, Verrous, Profileur, Architectures Multicoeurs

# 1 Introduction

Today, Java is regularly used to implement large multi-threaded server-class applications such as databases and web servers, where responsiveness is critical for a good user experience. Such server applications are designed around the use of shared data, that are accessed within critical sections, protected by locks, to ensure consistency. However, the use of locks decreases parallelism for the duration of the critical sections, and thus hampers scalability [3]. Restoring some of this parallelism requires careful and time-consuming optimization of the locks and critical sections that hamper the most the parallelism [8, 26]. Identifying these critical sections requires profiling the running application.

Profiling a Java server-class application is challenging, due to the long running time of such applications and the range of run-time conditions that can arise. A number of Java profilers provide lock profiling, reporting on the average contention for each lock over the entire application execution in terms of a variety of metrics. The metrics used by these profilers, however, do not always highlight the critical sections that have the highest impact on scalability. Furthermore, by reporting only an average over the entire application execution, these lock profilers are not able to identify local variations due to the properties of the different phases of the application. Localized contention within a single phase may harm responsiveness, but be masked in the profiling results by the long overall execution time.

To address these issues, there is a need for a profiler with the following properties: i) the profiler must measure the *critical section pressure* (CSP) of each lock, which we define as the ratio between the blocking time, in which threads are unable to enter critical sections because the lock is held by another thread, and the overall amount of computational resources occupied by the application, and ii) the profiler should recompute this metric periodically, to be sensitive to the different properties of the phases of the application. A high CSP may furthermore result from specific situations that are difficult for the developer to simulate exhaustively or that only arise on specific architectures that are not available to the developer. There is thus a further need for an *in-vivo* profiler that is able to detect phases with a high CSP when the application is run by users. However, current lock profilers for Java incur a substantial overhead, making their use only acceptable in an *in-vitro* development setting.

These issues are illustrated by a problem that was reported two years ago in version 1.0.0 of the distributed NoSQL database Cassandra [25].[1] Under a specific setting, with three Cassandra nodes and a replication factor of three, when a node crashes, the latency of Cassandra is multiplied by twenty. This slowdown is caused by a contended lock used in the implementation of *hinted handoff*[2], by which live nodes record their transactions for the purpose of later replay by the crashed node. First, developers seem not to have considered testing this specific scenario, or they tested it but were not able to cause the problem. Moreover, even if the scenario was by chance executed, other profilers would be unable to identify the cause of the bottleneck if the scenario was activated during a long run that hides the contention phase.

In this paper, we propose Free Lunch, a new lock profiler especially designed to identify *phases* of high CSP *in-vivo*. In order to identify such phases, Free Lunch periodically computes the CSP for each lock over a previous time interval. The CSP is based on computing the percentage of time spent by the application in acquiring the lock, which directly indicates the maximal theoretical improvement that the developer can expect by optimizing the lock's critical sections. When the CSP of a lock reaches a threshold, Free Lunch reports back to developers the identity of the lock, along with information to reproduce the issue, just as applications and operating systems now commonly report back to developers about crashes and other unexpected situations [14].

[1] See `https://issues.apache.org/jira/browse/CASSANDRA-3386`.

[2] `http://wiki.apache.org/cassandra/HintedHandoff`

In order to achieve our goal of *in-vivo* profiling, Free Lunch must incur little overhead. We have evaluated the main causes of performance overhead of other Java lock profilers. Our evaluation shows that the main bottleneck is that they use an internal lock, which becomes contended at high core count. To reduce this overhead, Free Lunch leverages the internal lock structures of the Java Virtual Machine (JVM), which are already thread-safe, and injects the process of periodically computing the CSP values into the JVM's existing periodic lock management operations.

We have implemented Free Lunch in the Hotspot 7 JVM. Free Lunch only requires modifying 420 lines of code, mainly in the locking subsystem, suggesting that it should be easy to implement in another JVM. We have evaluated Free Lunch on a 48-core AMD Magny-Cours machine in terms of both the performance penalty and the usefulness of the profiling results. Our results are as follows:

- We have found that the lock contention metrics used by the existing Java lock profilers MSDK [29], the Java Lock Monitor [28], the Java Lock Analyser, IBM Health Center [16], HPROF [17], JProfiler [22] and Yourkit [37] are inappropriate to indicate the performance increase that a developer can expect to achieve by optimizing the application's critical sections.
- Free Lunch makes it possible to immediately detect a previously unreported phase with a high CSP in the log replay subsystem of Cassandra. This issue is triggered under a specific scenario and only during a phase of the run, which makes it difficult to detect with current profilers.
- Free Lunch makes it possible to identify four locks with high CSP in four standard benchmark applications. Based on these results, we have improved the performance of one of these applications (Xalan) by 15% by changing only a single line of code. For the other applications, the information returned by Free Lunch helped us verify that the locking behavior could not be improved.
- On the DaCapo benchmark suite [5], the SPECjvm2008 benchmark suite [34] and the SPECjbb2005 benchmark [33] we find that there is no application for which the performance overhead of Free Lunch is greater than 6%. This result shows that a CSP profiler can have an acceptable performance impact for *in-vivo* profiling.
- The profilers HPROF [17], JProfiler [22], Yourkit [37] and MSDK [29] on the same set of benchmarks incur a performance overhead of up to 14 times, 9 times, 7.6 times and 35 times, respectively, making them unacceptable for *in-vivo profiling*.

The rest of this paper is organized as follows. Section 2 presents how synchronization is implemented in JVMs and the state-of-the-art in Java lock profiling. Section 3 presents the design of Free Lunch and Section 4 presents its implementation. We evaluate Free Lunch in Section 5. Section 6 presents related work and Section 7 concludes.

## 2 Background

In this section, we first describe the implementation of synchronization in modern JVMs, focusing on Hotspot 7. The same implementation strategy is used in other modern JVMs, such as Jikes RVM [1] and VMKit [13]. Free Lunch leverages this implementation to perform profiling efficiently. We then present the state-of-the-art Java profilers that are oriented towards measuring lock contention. We highlight some of their design decisions that induce a degree of

overhead that makes them unacceptable for *in-vivo* profiling. Finally, we discuss the limitations of contention metrics that they use.

## 2.1 Locking in the Hotspot 7 JVMs

In Java, each object has an associated *monitor* [15], comprising a lock and a condition variable. As typically only a few Java objects are used for synchronization, Hotspot 7 includes an optimization that minimizes the monitors' memory consumption [4]. This optimization is based on the following observations: (i) when no thread is blocked while waiting for the lock of the monitor, there is no need for a queue of blocked threads, and (ii) when no thread is waiting on the condition variable of the monitor, there is no need for a queue of waiting threads.

When both conditions hold, we say that the monitor is in *flat mode* (see Figure 1). In this case, the monitor is considered to be not contended and the Hotspot 7 JVM represents the monitor as only a few bits in the Java object header. These bits indicate whether the monitor is in flat mode and, if so, whether its lock is held. The monitor becomes contended when a thread tries to acquire the monitor lock while it is already held by another thread, or when a thread starts waiting on the monitor condition variable. In either case, the Hotspot 7 JVM *inflates* the monitor, so that it now contains thread queues. The same bits in the Java object header are then used to reference the inflated monitor structure.

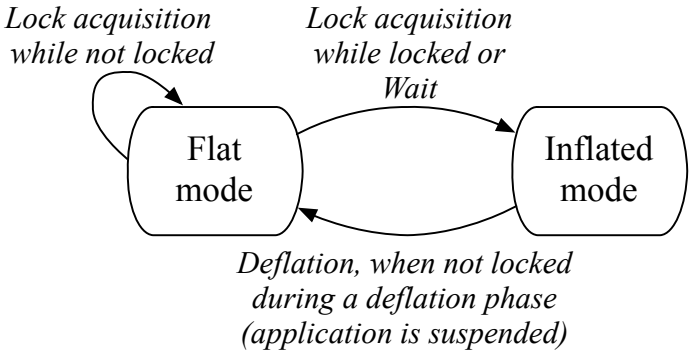



Figure 1: Transitions between flat and inflated mode.

If an inflated monitor becomes not contended because it has no waiting threads for either the lock or the condition variable, the Hotspot 7 JVM eventually *deflates* the monitor into flat mode. During deflation, the Hotspot 7 JVM has to prevent concurrent accesses from the application to the inflated monitor structure. For this reason, the Hotspot 7 JVM deflates a monitor only when the application is suspended. Hotspot 7 exploits the fact that it already regularly suspends all the threads in order to collect the memory, deoptimize a code or redefine a class [32]. Hotspot 7 leverages this synchronization to perform a deflation cycle each time all the threads are suspended. During a deflation cycle, Hotspot 7 inspects all the inflated monitors. If the monitor is not contended at this time, Hotspot 7 deflates it into flat mode.

## 2.2 Performance issues with existing profilers

Before designing Free Lunch, we studied HPROF [17], Yourkit [37], JProfiler [22], and MSDK [29], four commonly used general-purpose profilers for Java that provide lock profiling. These profilers

have been designed with the goals of portability and independence from the targeted JVM. Thus, each of them is implemented as a dynamic shared library that is loaded on start up of the JVM, and requires no modifications to the JVM or the application.

To allow the profiler to determine the amount of time spent to acquire a lock, these existing profilers use JVMTI [23], a standard JVM-independent Java interface that provides data about the state of the JVM. Each of the profilers registers two event handlers through the JVMTI API: one that is called before a thread is suspended because it tries to acquire a locked lock, and another that is called after the thread has acquired the lock. Using these event handlers, the profilers compute either the time spent in acquiring each lock or the number of lock acquisition failures.

To identify the cause of overhead of these profilers, we have evaluated HPROF, which we have found to introduce the lowest overhead on most of our test applications (see Section 5.1). Our evaluation shows that half of the overhead is due to the use of an external map associating Java locks to their profiling data, and that the other half is due to interaction with the JVM. We describe these issues in more detail below.

**External map use.** Being independent of the JVM implementation implies that HPROF needs to maintain a map to associate each object with its profiling data. Each time a lock-related event is fired, the profiler uses this map to find the profiling data associated with the object indicated by the event. In the case of the second event, which is fired during a critical section, this computation further delays all of the waiting threads, thus multiplying the performance impact by the number of these threads. Moreover, a lock is needed in the implementation of this map to prevent multiple threads from accessing it concurrently. Managing this lock also slows down the application.

**JVM interaction.** When the JVM terminates, HPROF has to dump a coherent view of the collected data. In the case of a general-purpose profiler, some event handlers may collect multiple types of information. To ensure that the dumped information is consistent, HPROF must ensure that no handler is executing while the dump is being prepared. HPROF addresses this issue by continuously keeping track of how many threads are currently executing any JVMTI event handler, and by only dumping the profiling data when this counter is zero. HPROF protects this counter with a single lock that is acquired twice on each fired event, once to increment the counter and once to decrement it, which further slows down the application.

## 2.3 Lock contention metrics

Several metrics have been proposed for measuring lock contention. In the rest of this section, we present two application scenarios that we use to analyze these metrics. We demonstrate that each of them does not satisfy our goal for at least one of the two scenarios. Table 1 presents the metrics and the seven profilers of which we are aware, and Table 2 summarizes our analysis.

### 2.3.1 Scenarios.

Our first scenario is the ping-pong scenario shown in Figure 2, in which two threads execute in mutual exclusion. Each thread executes an infinite loop. During each iteration, a thread acquires a lock, executes a processing function, and releases the lock. If this scenario is executed on a two-core machine, only half of the capacity of the machine is used because at any given time, only one thread can make progress. We would like a metric to produce a high value in this

Table 1: Lock contention metrics (top row divided by left column)

| | # of failed acquisitions | CS time of a lock | Acquiring time |
|---|---|---|---|
| Elapsed time | Java Lock Monitor, Java Lock Analyzer, MSDK | Java Lock Monitor, Java Lock Analyzer, IBM Health Center | |
| # of acquisitions | IBM Health Center, Mutrace | | |
| CS time of all locks | | HPROF | |
| Nothing | | | JProfiler, Yourkit |

| Contention metric | Scenario | | Profilers |
|---|---|---|---|
| | ping-pong | fork-join | |
| # failed acquisitions/ elapsed time | ☹ | ☺ | MSDK, Java Lock Monitor Java Lock Analyser |
| # failed acquisitions/ # acquisition | ☺ | ☹ | IBM Health Center, Mutrace |
| CS time of a lock/ elapsed time | ☹ | ☺ | Java Lock Monitor, Java Lock Analyser, IBM Health Center |
| CS time of a lock/ CS time of all the locks | ☹ | ☹ | HPROF |
| Acquiring time/Elapsed time | ☹ | ☹ | JProfiler, Yourkit |

Table 2: Analysis of contention metrics.

scenario, because if it is possible to move some of the code out of the critical sections, then the developer can expect a large improvement.

We also consider the fork-join scenario shown in Figure 3, in which a master thread distributes work to worker threads and waits for the result. The scenario involves the monitor methods `wait()`, which waits on a condition variable, and `notifyAll()`, which wakes all threads waiting on a condition variable. Both methods must be called with the monitor lock held. The `wait()` method additionally releases the lock before suspending the thread, and reacquires the lock when the thread is reawakened.

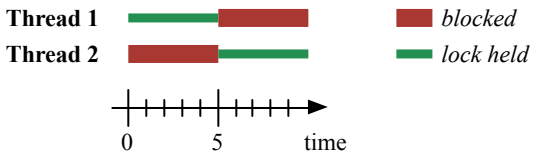



Figure 2: A ping-pong scenario.

In the fork-join scenario, the workers alternate between performing processing in parallel (narrow solid lines) and waiting to be awakened by the master (red and green thick lines and dashed lines). Initially, the workers are waiting, having previously invoked the `wait()` method and the master holds the lock. At time 0, the master wakes the workers using `notifyAll()`. Each worker receives the notification (time 1), but to continue must be able to first reacquire

the lock and then leave the critical section containing its `wait()` call. This leads to a cascade of blocked workers, over times 1-5: The first worker has to wait for the master to release the lock, which it does by performing a `wait`, at time 2. The second worker then has to additionally wait for the first worker to exit the critical section and release the lock (time 3), etc. The workers then perform their processing, in parallel, over times 3-11. When each worker completes its processing, it again enters the critical section, at times 8, 9, 10, and 11, respectively, to be able to invoke `wait` (time 9-14), to wait for the master. This entails acquiring the lock, and then immediately releasing it via the `wait()` method. Finally, when the fourth worker completes its processing (time 11), it acquires the lock and uses `notifyAll()` to wake the master (time 12). At this point, the master must reacquire the lock, which is currently held by the fourth worker. The fourth worker releases the lock when it invokes `wait()` (time 14), unblocking the master and starting the entire scenario again.

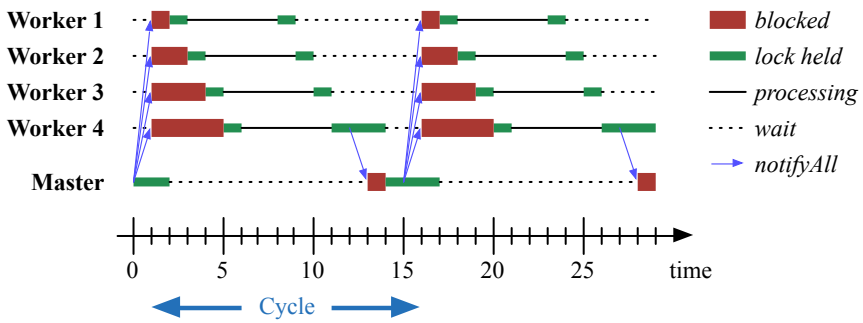



Figure 3: A fork-join scenario.

In this scenario, all of the workers are recurrently blocked on the same lock, while trying to exit the `wait()` method. As few instructions are involved, however, the block time is likely to be very short. The potential benefit of optimizing this synchronization depends on the relationship between the blocked time and the processing time. If the processing time of the workers is large, optimizing the synchronizations between the workers and the master is useless, while it becomes beneficial if the processing time decreases. A metric should thus reflect this trade-off.

### 2.3.2 Metrics based on the number of failed acquisitions.

Several profilers rely on metrics based on the number of failed acquisitions, i.e., the number of times where the lock acquisition method detects that the lock is already held. The idea behind these metrics is that the probability of a lock acquisition failing increases with the contention.

MSDK [29], Java Lock Monitor from the Performance Inspector suite [28] and Java Lock Analyser [20] report the number of failed acquisitions per time unit. We consider this metric with respect to the ping-pong scenario, on which we expect a high value. In this scenario, after each round of processing, which takes place with the lock held, both threads are trying to acquire the lock and one of them will fail. The number of fails per time unit is thus equal to one divided by the time of the processing function (the narrow green rectangle in Figure 2). If the processing function takes a lot of time, the number of fails per time unit will be small and will not reflect the actual contention.

IBM Health Center [16] for Java applications, and Mutrace [30] for C applications report the number of failed acquisitions divided by the total number of acquisitions. With the fork-join scenario (see Figure 3), the total number of acquisition fails divided by the number of acquisitions is equal to 5/9, with 4 failed acquisitions by the workers at time 2, 4 successful acquisitions by the workers at times 8, 9, 10 and 11, respectively, and 1 failed acquisition by the master at time 13. This value is a constant and is not related to the actual contention, which can be high or low, depending on the relationship between the time for synchronization and the time for the rest of the processing in a cycle.

#### 2.3.3 Metrics based on the critical section time.

Other widely used metrics are based on the time spent by the application in the critical sections associated with a lock. The idea behind these metrics is that a contended application will spend most of its time in critical section.

Java Lock Monitor [28], Java Lock Analyser [20] and IBM Health Center [16] use this metric. They report the time spent by the application in the critical sections associated to a lock divided by the elapsed time of the application. On the ping-pong scenario (see Figure 2), 100% of the application time is spent in critical sections (narrow green rectangles), meaning that the application has a high contention, as we expect. Now let us consider the same scenario, but where there is only one thread. This thread will never need to block to enter the critical section, in which it will spend 100% of its time. The metric would thus still report a contention of 100%. However, with this new scenario, the lock is not contended because the lock never prevents a thread from executing.

HPROF [17] reports the time spent by the application in each lock's critical sections divided by the total time spent by the application in any critical section. On our degenerate one-thread variant of the ping-pong scenario, the metric will again report that 100% of the time is spent in the only lock, while the application does not suffer from contention.

#### 2.3.4 Metrics based on the acquiring time.

JProfiler [22] and Yourkit [37] report the time spent by the application in acquiring each lock. To provide a meaningful measure of contention, this metric would have to be related to the overall execution time of the application. However, JProfiler and Yourkit only report the elapsed time of the application, which does not take into account the fact that multiple threads execute. For example, let us consider an application with ten threads that execute during one minute and for which each thread spends half of its time in acquiring the lock. JProfiler and Yourkit report that the application spends five minutes in lock acquisitions, while it executes for only one minute. Without knowing the number of threads, which can evolve during the execution, it is not possible to decide whether the lock is a bottleneck.

## 3 Free Lunch Design

The goal of Free Lunch is to identify the locks exhibiting the highest CSP and to regularly assess their CSP over a preceding time interval. We refer to this interval as the *measurement interval*. We now describe our design decisions with respect to the definition of our contention metric, the duration of the measurement interval, and the information that Free Lunch reports to the developer.

### 3.1 Free Lunch metric

Free Lunch defines the CSP of a lock as the ratio of i) the time spent by threads in acquiring the lock and ii) the cumulated running time of these threads. Thus, like JProfiler [22] and Yourkit [37], Free Lunch uses the acquiring time, but Free Lunch adjusts it to a notion of thread execution duration, allowing the developer to understand the performance penalty caused by the critical section. By design, the Free Lunch metric reflects the performance improvement that is possible by reducing the amount of time that the application spends in critical sections. Indeed, if the percentage of time spent in acquiring a given section becomes large, it means that the threads of the application are not able to execute for long periods of time because they are blocked.

Let us consider the ping-pong scenario of Figure 2 in terms of the Free Lunch metric. In this scenario, Free Lunch reports a CSP of 50% because each thread is blocked 50% of the time (large red rectangles). This CSP measurement is satisfactory because it indicates that with an ideal optimization, a developer could divide nearly up by two the processing time by moving the processing outside any critical section. Moreover, if we consider the variant of this scenario where only a single thread runs and holds the lock, Free Lunch will report that the application spend 0% of its time in lock acquisition, reflecting the fact that the lock is not a bottleneck.

In the fork-join scenario (see Figure 3), Free Lunch will report a CSP equal to the sum of the times spent while blocked (large red rectangles) divided by the sum of the running time of the threads. Contrarily to the metric that divides the number of fails by the number of acquisitions, the Free Lunch metric increases when the processing time decreases, thus indicating that a developer could optimize its application by optimizing the synchronization between the workers and the master.

### 3.2 Measurement interval

In order to identify the phases of high CSP of an application, Free Lunch computes the CSP of each lock over a measurement interval. Calibrating the duration of the measurement interval has to take two contradictory constraints into account. On one hand, the measurement interval has to be small enough to identify the phases of an application. If the measurement interval is large as compared to the duration of a phase of the application in which there is a high CSP, the measured CSP becomes negligible and Free Lunch is unable to identify the high CSP phase. On the other hand, if the measurement interval is too small, few blocked threads during a small interval can make the CSP reaches a high level, while there is little pressure on the critical section. Due to these variations, Free Lunch will identify a lot of phases of very high CSP, hiding the actual high CSP phases with a lot of false positive reports.

We have tested a range of intervals on the Xalan application from the DaCapo benchmark suite. This application is an XSLT parser transforming XML documents into HTML. This application exhibits a high CSP phase at the end of the execution caused by a lot of synchronized accesses to a hash table. Figure 4 reports the evolution of the CSP over time. When the measurement interval is 5ms, the CSP varies a lot between successive measurement points. In this case, the lock bounces back and forth from being contended (high points) to not being contended (low points). At the other extreme, when the measurement interval is approximately equal to the execution time (13s), the CSP is averaged over the whole run, hiding the phases. With a measurement interval of 1s, we can observe that (i) the application has a high CSP during the second half of the run with a value that reaches 15%, (ii) the CSP remains relatively stable between two measurement intervals.

Based on the above experiments, we conclude that 1s is a good compromise, as this measurement time is large enough to stabilize the CSP value. Moreover, if a high CSP phase is shorter

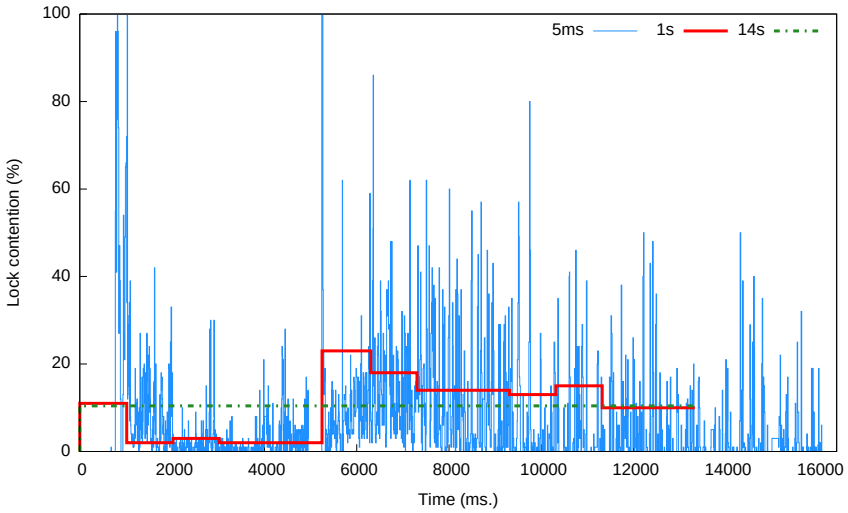



Figure 4: CSP depending on the minimal measurement interval for the Xalan benchmark.

than 1s, it is likely that the user will not notice the degradation in responsiveness.

### 3.3 Free Lunch reports

To further help developers to identify the source of high CSP, Free Lunch reports not only the identity of the affected locks, but also, for each lock, an execution path that led to a contended acquisition. Free Lunch obtains this information by traversing the runtime stack. Because traversing the runtime stack is expensive, Free Lunch only records the call stack that leads to the execution of the acquire operation that causes the monitor to be inflated for the first time. Previous work [2] and our experience in analyzing Java programs, described in Section 5.3, shows that only a single call stack is generally sufficient to understand the cause of contention.

## 4 Free Lunch implementation

This section presents the implementation details of Free Lunch in the Hotspot 7 JVM for an x86 architecture. We first describe how Free Lunch measures the different times required to compute the CSP. Then, we present how Free Lunch efficiently collects the relevant information. Finally, we present some limitations of our implementation.

### 4.1 Time measurement

Free Lunch has to compute the cumulated time spent by all the threads on acquiring each lock and the cumulated running time of all the threads (see Figure 5). Below, we describe how Free Lunch computes these times.

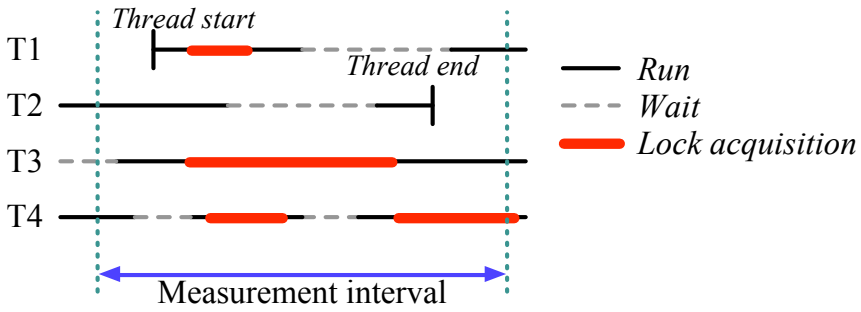



Figure 5: Time periods relevant to the contention computation.

#### 4.1.1 Acquiring time.

The acquiring time is the time spent by the thread in acquiring the lock, in the case that the lock is already held by another thread. The acquiring time is computed on a per lock basis. For this, we have modified the JVM lock acquisition method to record the time before and the time after the acquisition. A challenge is then where to store this information for further computation. Indeed, one of the causes of the high runtime overhead of HPROF is the use of a map that associates each Java object to its profiling data. Free Lunch avoids this cost by directly recording the acquiring time in a field added to the monitor structure of the JVM. As Free Lunch records this elapsed time while the lock is held, it avoids the need to introduce another lock to prevent concurrent access to the monitor structure.

To accurately obtain a representation of the current time, Free Lunch uses the x86 instruction `rdtsc`, which retrieves the number of elapsed cycles since the last processor restart. As most x86 architectures support instruction reordering, there is, in principle, a danger that the order of `rdtsc` and the lock acquisition operation could be interchanged. To address this issue, general-purpose profilers, such as PAPI [11], that use `rdtsc` must also introduce an additional costly instruction to prevent reordering. Fortunately, a Java lock acquisition triggers a full memory barrier [27], across which the x86 architecture never reorders instructions, and thus such a full memory barrier instruction is not needed. Obtaining the current time when requesting a lock requires the execution of four x86 assembly instructions including `rdtsc` and data formating. Obtaining the current time after acquiring the lock, computing the elapsed lock acquisition time, and storing it in the lock structure requires the execution of seven x86 assembly instructions.

A potential limitation of our strategy of storing the acquiring time in the monitor structure is that this structure is only present for inflated monitors. Free Lunch thus collects no information when the monitor is deflated. By definition, however a flat lock is not contended, and thus not counting the contention in this case does not change the result.

#### 4.1.2 Running time

So as to compute the cumulated running time, we have chosen to not consider the time when a thread does not have work to perform, that is in Java, when a thread waits on a condition variable. This waiting time is not essential to the computation of the application and including it would drastically reduce the CSP, making the identification of high CSP phases difficult. In practice, there are two ways for a thread to wait on a condition variable: either by calling the

`wait()` function on a monitor, or by calling the `park()` function from the `sun.misc.Unsafe` class. To exclude the waiting times, Free Lunch records the current time just before and after a call to one of these functions in a thread local variable. At the end of the measurement interval, Free Lunch sums these waiting times and subtracts the result from the elapsed time of the threads.

## 4.2 CSP computation

Computing CSP is a costly operation. First, Free Lunch has to visit all of the threads to sum up their running times. Second, Free Lunch has to visit all of the monitor structures to retrieve the lock acquiring time. For each lock, the CSP is then computed by dividing the acquiring time by the sun of the running times.

To avoid the extra cost of a new visit to each of the threads and monitors, Free Lunch leverages the visits already performed during the optimized lock algorithm presented in Section 2. The JVM regularly inspects each of the locks to possibly deflate them, and this inspection requires that all Java application threads be suspended. Since, suspending the threads already requires a full traversal of the threads, Free Lunch leverages this traversal to compute the accumulated waiting times. Free Lunch also leverages the traversal of all the monitors performed during the deflation phase to compute their CSP.

## 4.3 Limitations of our implementation

We have noted that implementing Free Lunch inside the JVM makes it possible to avoid the use of a costly map to associate monitor addresses to profiling data. Storing profiling data inside the monitor data structure in Hotspot 7, however, is not completely reliable, because deflation can break the association between a Java object and its monitor structure at any time, causing the data to be lost. Thus, Free Lunch manages a map that associates every Java object memory address to the associated monitor. During deflation, Free Lunch adds the current monitor to that map. When the lock becomes contended again, the inflation mechanism looks into this map to check if a monitor was previously associated with the Java object being inflated. As compared to the map used in solutions that are independent of the JVM, this map is only accessed during inflation and deflation, which are typically far less frequent than lock acquisition.

Our solution to keep the association between a Java object memory address and its associated monitor is, however, not sufficient in the case of a copying collector [21]. Such a collector can move the object to a different address while the monitor is deflated. In this case, Free Lunch will be unable to find the old monitor. A solution could be to update the map when an object is copied during the collection. We have not implemented this solution because we think that it would lead to a huge slowdown of the garbage collector, as every object would have to be checked.

We have, however, observed that having a deflation of the monitor followed by first a copy of the object and then a new inflation of the monitor within a single phase is extremely rare in practice. Indeed, a monitor is deflated when it is no longer contended and the deflation will mostly happen between high CSP phases. As a consequence, the identification of a high CSP phase is not altered by this phenomenon. If several high CSP phases are associated with a same lock, the developer will receive multiple reports, and the reports will not indicate that they all relate to the same lock. We do not think that this is an issue, because the developer will easily see from the code that all of the reports relate to a single lock.

# 5 Experiments

We now evaluate Free Lunch. All of our experiments were performed on a 48-core 2.1GHz AMD Magny-Cours machine having 256GB of RAM. The system runs a Linux 3.0.0 64-bit kernel from Ubuntu 10.10. We use OpenJDK version 7 for HPROF, FreeLunch, Yourkit and JProfiler , and IBM J9 1.7 for MSDK. We test Cassandra 0.7.0 [25], 11 applications from the DaCapo benchmark suite [5], 19 applications from the SPECjvm2008 [34] benchmark suite, and SPECjbb2005 [33].

First, we study HPROF in detail, in order to better understand the cause of its overhead. Then, we study the overhead of Free Lunch. Finally, we analyze the results reported by Free Lunch.

## 5.1 Overhead of existing-profiler mechanisms

In Section 2.2, we observed that profilers that are designed to be independent of the implementation of the JVM result in an overhead, due to the cost of an external map and JVM interaction. Here we assess these costs in the context of HPROF. We have found that HPROF has the lowest overhead of the non-sampling profilers considered, and it is the only profiler whose source code is available, making it possible to correlate the observed performance to implementation features.

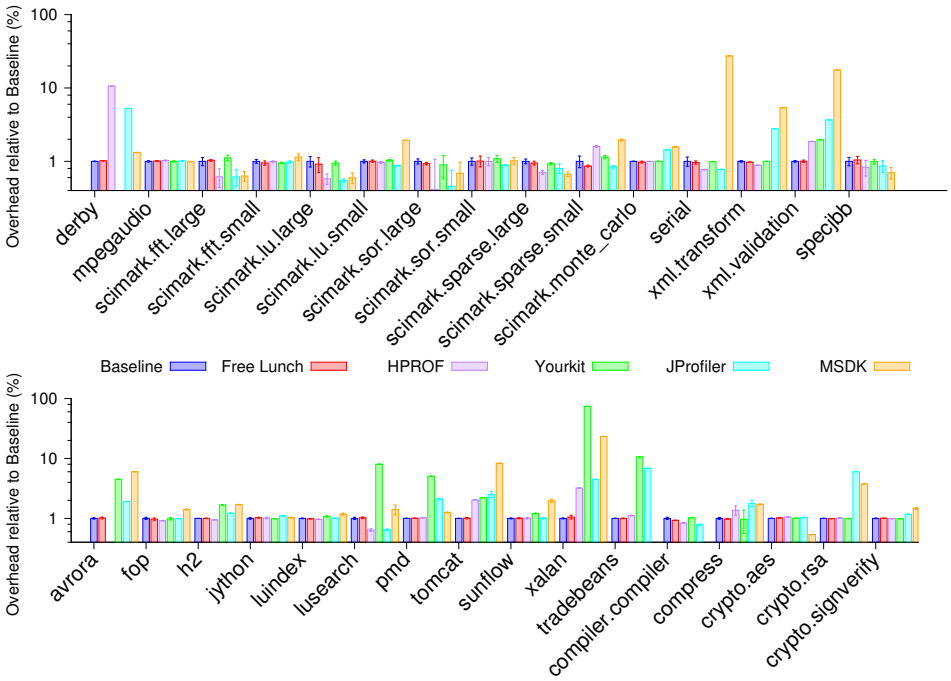
We first evaluate profiler overhead in terms of the number of cores and threads. In order to evaluate the impact of the number of threads on the overhead, we compared HPROF to Hotspot, without profiling, on Xalan in two configurations: 2 threads on 2 cores, and 48 threads on 2 cores. In both cases, the overhead caused by the profiler is around 1%, showing that the number of threads only has a marginal impact on profiler performance. Then, in order to evaluate the impact of the number of cores on the overhead, we evaluated Xalan with 48 threads on 48 cores. In this case, Xalan runs 3.9 times slower. These results thus suggest that the overhead of HPROF mainly depends on the number of cores.

To better understand the cause of this overhead at high core count, we consider the cost of accessing the map data structure during profiling. For this, we completely removed all accesses to the map, and thus the associated synchronizations, for the DaCapo Xalan benchmark. With 48 threads on 48 cores, we found that the overhead incurred by the map represents 42% of the total overhead of HPROF. Finally, we consider the cost of maintaining a thread counter protected by a lock to keep track of how many threads are currently executing any JVMTI event handler. When removing the lock, we found that the overhead incurred by this counter represents 58% of the overhead of HPROF.

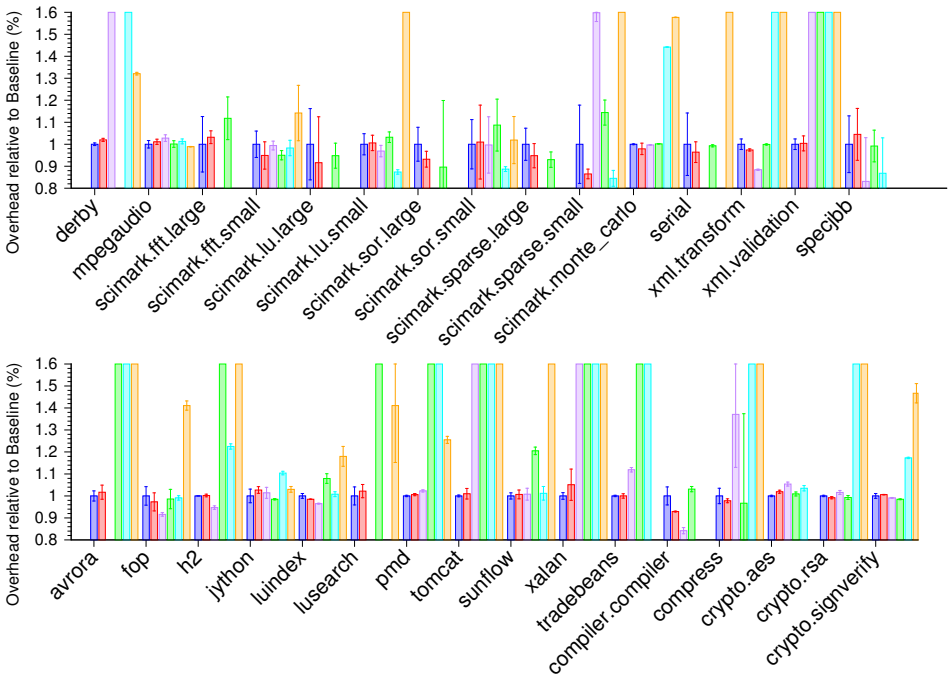
## 5.2 Free Lunch overhead

We first compare the overhead of Free Lunch to that of HPROF, Yourkit, JProfiler and MSDK using their lock profiling mode. For DaCapo, we run each application 5 times with 10 iterations, and take the average execution time of the last iteration on each run. For SPECjvm2008, we set up each application to run a warmup of 120s followed by 10 iterations of 240s each. For SPECjbb2005, we create an experiment that uses 48 warehouses and runs for 240s with a warmup of 120s. For SPECjvm2008 and SPECjbb2005, we report the average rate of operations completed per minute.

Figure 6 presents the overhead incurred by each of the profilers, as compared to the baseline JVM with no profiling (J9 for MSDK and Hotspot for the others). Results are presented in two ways in order to account for the wide variations. Figure 6.a presents the complete results, on a logarithmic scale, while Figure 6.b focuses on the case between 20% speedup (0.8) and 60% slowdown (1.6).




(a) Overhead on execution time with a logarithmic scale.




(b) Overhead on execution time between 80% and 160% (zoom of (a)).

Figure 6: Overhead on execution time compared to baseline.

The top graph shows that HPROF, Yourkit, JProfiler and MSDK can degrade application performance by an order of magnitude. The overhead of HPROF can be up to 14 times, that of Yourkit up to 9 times, that of JProfiler up to 7.6 times and that of MSDK up to 35 times. The bottom graph shows that for all applications, the average overhead of Free Lunch is always below 6%.

We furthermore observe that the use of HPROF, Yourkit, JProfiler, and MSDK makes it impossible to execute some of the benchmarks: Compiler.compiler and Tradebeans do not run with MSDK, Derby does not run with Yourkit and Avrora does not run with HPROF.

## 5.3 Analysis of lock CSP

Table 3 reports the locks that have the highest CSP for each application, with a measurement interval equal to the elapsed time. We include only applications for which there is at least one lock for which the application spends at least 1% of its overall execution time on lock acquisition. In the remainder of the section, we focus our analysis on the five applications with the highest percentage of time spent in acquiring locks and analyze in detail the evolution of CSP during a run. Thanks to the report provided by Free Lunch, we also describe the causes of the high CSP.

Table 3: Locks with the highest CSP

| Benchmark | Java class of the contended object | % of total run time spent in lock acquisition |
|---|---|---|
| H2 | org.h2.engine.Database | 44.0% |
| Pmd | org.dacapo.harness.DacapoClassLoader | 40.0% |
| Sunflow | org.sunflow.core.Geometry | 8.8% |
| Xalan | java.util.Hashtable | 7.5% |
| Avrora | java.lang.Class | 7.3% |
| Tradebeans | org.h2.engine.Database | 4.6% |
| Crypto.rsa | java.security.SecureRandom | 1.7% |
| Tomcat | org.apache.jasper.servlet.JspServletWrapper | 1.0% |

H2 is an in-memory database. On average, H2 spends 44.0% of its time acquiring a lock associated with a `org.h2.Database` object. H2 uses this lock to ensure that client requests are processed sequentially; thus, the more clients send requests to the database, the more clients try to acquire the lock. Figure 7.a presents the lock CSP throughout the execution of H2. We see distinctly 3 phases. The first phase (from 0 to 15 seconds) presents no CSP at all: in this phase the main thread of the application populates the database, thus no CSP occurs for accessing the database. The second phase (from 15 to 62 seconds) shows a CSP between 78% and 83%: clients are sending requests to the database, thus inducing contention on the database lock. The CSP decreases at the end of the phase when clients have finished their requests to the database. The purpose of the last phase (from 62 seconds to the end) is to revert the database back to its original state, which is again done only by the main thread and thus induces no CSP. This application is inherently not scalable because requests are processed sequentially. Deep modifications would be required to improve performance.

Pmd is a source code analyzer. On average, it spends 40.0% of its time acquiring a lock associated with the `org.dacapo.harness.DacapoClassLoader`. This class is used to load new classes during execution. Figure 7.b presents the CSP throughout the execution of Pmd. The high CSP phase begins at 2s and terminates at 8s, while the application terminates at 10s. During the high CSP phase, Pmd stresses the class loader because all the threads are trying to

load the same classes. Removing this bottleneck is likely to be hard because the classes have to be loaded serially.

Sunflow is an image rendering application. On average, it spends 8.8% of its time acquiring a lock associated with a `org.sunflow.core.Geometry` object. Figure 7.c presents the CSP throughout the execution of Sunflow. This curve shows a moderate CSP peak at the beginning of the execution. This is due to the tesselation of 3D objects which must be done in mutual exclusion. Since the number of 3D objects is small as compared to the number of threads, many threads must block, waiting for the tesselation to complete, therefore delaying the image rendering. This result suggests that the working set used in the DaCapo benchmark is not adequate for a machine with a large number of cores because Sunflow creates as many threads as cores.

Xalan is a XSLT parser transforming XML documents into HTML. On average, Free Lunch reports that Xalan spends 7.5% of its time trying to acquire the lock associated with a single `java.util.Hashtable` object. The class `java.util.Hashtable` uses a lock to ensure mutual exclusion on each access to the hashtable, leading to a bottleneck. Figure 7.d presents the percentage of CSP throughout the execution of Xalan. During the first phase (from 0 to 6 seconds) only one thread fills the hashtable, and therefore the CSP is low. However, during the second phase (from 6 seconds to the end of application), all the threads of the application are accessing the hashtable, increasing the CSP up to 17 %. This high CSP phase is hidden if we average the CSP during the whole run. We reimplemented the hash table using `java.util.concurrent.ConcurrentHashMap`, which does not rely on locks. This change required modifying a single line of code, and improved the baseline application execution time by 15%. This analysis shows that the information generated by Free Lunch can help developers in practice.

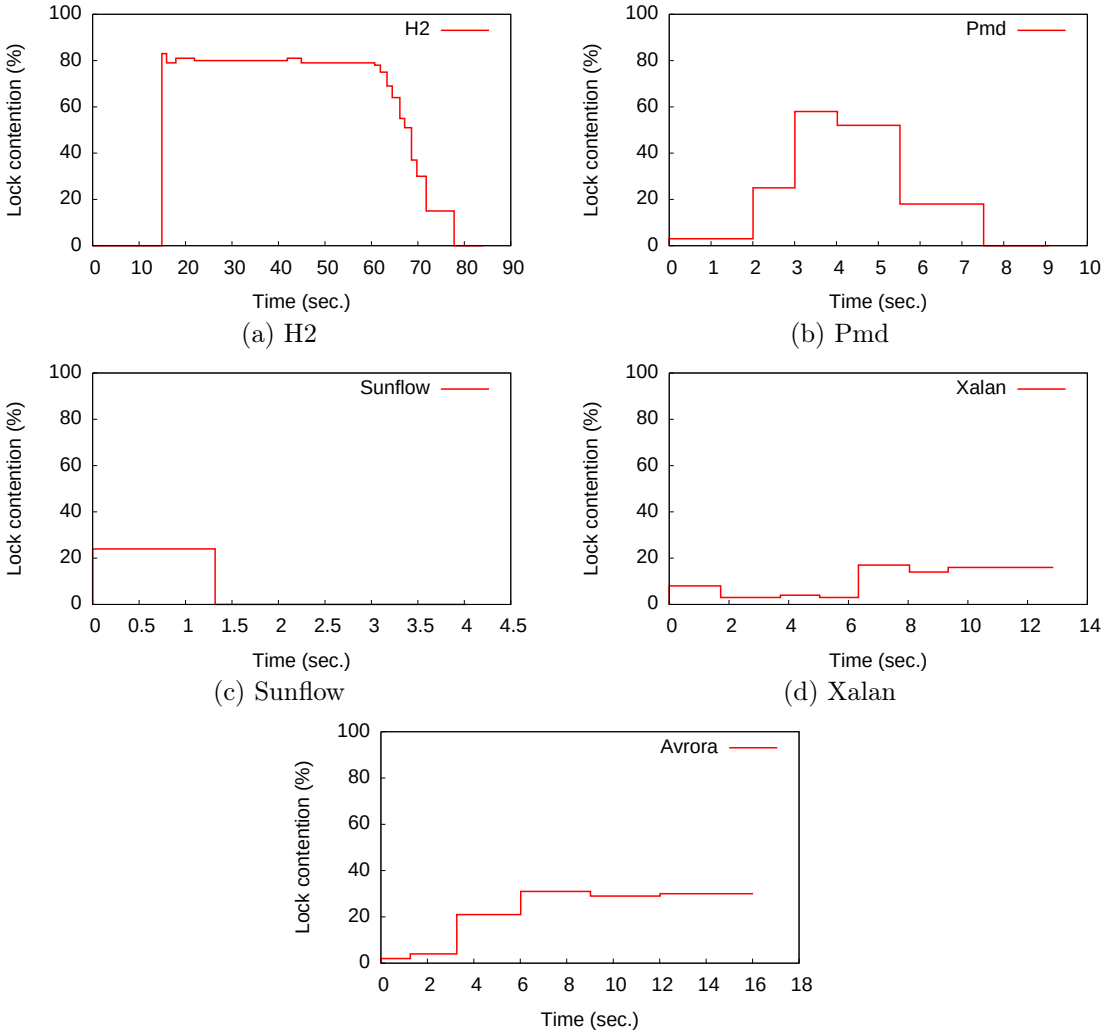
Avrora is a simulation and analysis framework. On average, it spends 7.3% of its time acquiring a lock associated with the `java.lang.Class` object. Avrora uses this lock to serialize its output to the terminal. Figure 7.e presents the CSP throughout the execution of Avrora. The graph distinctly shows the phase (from 3 seconds to the end of the application) where application threads write output results to a file. There seems to be no simple solution to remove this lock because interleaving outputs from different threads would lead to an inconsistent result.

## 5.4 Cassandra

Cassandra [25] is a distributed on-disk NoSQL database, with an architecture based on Google's BigTable [7] and Amazon's Dynamo [9] databases. It provides no single point of failure, and is meant to be scalable and highly available. Data are partitioned and replicated over the nodes. Durability in Cassandra is ensured by the use of a commit log where it records all the modifications. As exploring the whole commit log to answer a request is expensive, Cassandra also has a cache of the state of the data base. This cache is partially stored to disk and partially stored in memory. After a crash, a node has to rebuild this cache before answering client requests. For this purpose, it rebuilds the cache that was stored in memory by replaying the modifications from the commit log.

A Cassandra developer reported a lock contention issue in Cassandra 1.0.0.[3] During this phase, the latency was multiplied by twenty. The issue was observed on a configuration where the database is deployed on three nodes with a replication factor of three, and consistency is ensured by a quorum agreement of two replicas. No further information about the configuration is provided. As a result, we were unable to reproduce this problem.

[3]See `https://issues.apache.org/jira/browse/CASSANDRA-3385` and `https://issues.apache.org/jira/browse/CASSANDRA-3385`.




(a) H2 (b) Pmd

(c) Sunflow (d) Xalan

(e) Avrora

Figure 7: Evolution of lock CSP.

Although we were not able to reproduce this problem, we were able to detect a phase with a high CSP in Cassandra 0.7.0 thanks to Free Lunch. Using the configuration described above, we created a 23Gb database and then used the YCSB [31] benchmark to stress Cassandra with an update-heavy workload including 50% reads and 50% updates. After ten minutes, we simulated a crash by halting a node for 10 seconds and then simulated the recovery of the node by restarting it. During the replay of the commit log, which lasts for 90 seconds, FreeLunch reports a CSP of around 12% for one lock, which is comparable to that observed for Xalan, while the CSP for this lock is almost negligible otherwise. Coincidentally, the critical section involved was the same one that caused the previously reported problem in Cassandra 1.0.0. The fix in later versions solved both problems.

This experiment illustrates the difficulty of producing and reproducing CSP issues. Indeed, the particular tested scenario is complex to deploy and involves a server crash, which is relatively

unusual. For this reason, we think that the probability of encountering the issue during in-vitro testing is small, and thus in-vivo profiling is essential.

# 6 Related work

**Lock profilers.** Profilers that, like Free Lunch, instrument the code to continuously profile the application include HPROF [17], Yourkit [37], JProfiler [22], MSDK [29] and JLM (Java Lock Monitor), which is part of the Performance Inspector suite [28]. These profilers capture information about lock usage and rely either on the JVMTI API or the JVMPI API (the ancestor of JVMTI). They have to use an external map to associate Java locks to their profiling data. We have seen in the evaluation (Section 5) that using an external map drastically degrades application performance. Moreover, in Section 2 we have also seen that their metrics are not able to report a useful value on some synchronization patterns. Finally, these profilers report their metric at the end of the application, which hides phases.

For C applications, mutrace [30] and the profiler used in RCL [26] are also able to identify the most contended locks. As they do not modify the internal lock structure of the POSIX library, they also have to use a map to associate locks with their profiling data. They thus face the same issues as the previous profilers.

A second class of profilers relies on the sampling of some kinds of instructions. These include IBM Health Center [16] and the work of Inoue et al. [19], both for the IBM J9 JVM. When a sampling profiler observes a thread, it can at best know whether the thread is suspended or not, and, if it is not suspended, which instruction it is executing. However, a thread can be suspended for many reasons: because it is waiting on a condition variable, because it is blocked while waiting for a lock, because it is waiting for an I/O to complete or simply because it is not scheduled. Current sampling profilers cannot determine why a thread is suspended and thus cannot isolate the acquiring time, as we have done in Free Lunch. For this reason, a sampling profiler has to use a metric based either on the number of failed acquisitions or on the time spent in critical sections. But, as we have shown in Section 2, the two metrics proposed by IBM Health Center do not seem able to report the lock that hampers most the parallelism for some synchronization patterns. For example, we have evaluated Xalan with IBM Health Center and it did not report the contended lock associated with the hashtable that Free Lunch detects. Moreover, we have also evaluated IBM Health Center on IBM J9 JVM. Our results show that Health Center has an overhead that makes it suitable for *in-vivo* lock profiling. However, we noticed that J9 without profiling is at least 2 times slower than Hotspot 7 on 9 out of 31 benchmarks. On the Xalan benchmark, J9 is 7.3 times slower than Hotspot. These differences makes it difficult to compare a profiler that runs on J9 with a profiler that runs on Hotspot.

HaLock [18] is a hardware assisted lock profiler. It relies on a specific hardware component that tracks memory accesses in order to detect heavily used lock. This technique achieves low overhead but requires dedicated hardware.

Xian et al. [36] propose to dynamically detect lock contention induced by the OS on Java applications at runtime. The idea is to segregate threads that contend for the same lock on the same core and ensure that the lock owner run is allowed to run as long as it owns the lock. Therefore, it avoids lock contention induced by OS activities, such as thread preemption. This approach is complementary to ours because it focuses on lock contention induced by the OS, whereas Free Lunch focuses on lock contention induced by applications.

WAIT [2] is a tool that diagnoses various performance issues in running server-class applications. To measure lock usage, WAIT counts the number of threads blocked while acquiring lock. The rate of sampling is very low (1 samples every 1-2 minutes), suggesting that it is likely to

miss short lock usage phases like the one containing the performance bug found in Cassandra.

Lockmeter [6] is a tool that targets spinlock profiling for the Linux kernel. Like, e.g., Java Lock Monitor [28], Lockmeter reports the time spent in the critical section protected by a spinlock divided by the elapsed time. As shown in Section 2, this metric is not able to report a useful value on some synchronization patterns.

**Other profilers for parallel applications.** Bottle Graphs [12] is a profiling tool that is able to graphically illustrate the parallelism of an application. The degree of parallelism is mainly defined as the time where threads are not suspended divided by their execution time. Bottle Graphs is very useful in understanding whether the parallelism of the application could be enhanced and in identifying how each thread contributes to the processing, and thus it reports a macroscopic view of the parallelism of an application. Free Lunch is complementary to Bottle Graphs, as it is able to indicate whether a lack of parallelism comes from lock usage.

Kalibera et al. [24] define new concurrency metrics and analyse communication patterns of shared Java objects, and apply them to the DaCapo benchmarks [5]. They evaluate locking behaviour by counting the number of monitor acquisitions and the global locking rate of the application, along with the pattern by which these objects are accessed by threads. This work is complementary to ours, in that it gives a global view of shared-object behaviour whereas Free Lunch provides detailed information about CSP for each lock.

Limit [10] provides a lightweight interface to on-chip performance counters. Indeed, elapsed times obtained using `rdtsc` can be inaccurate when the number of threads exceeds the number of cores. Limit solves with issue by using a dedicated kernel module. The precision of Free Lunch could be improved in cases when the number of threads exceeds the number of cores by using Limit.

HPCToolkit [35] is a profiler designed for high performance computing. The authors define a new metric to attribute lock contention to the threads that are responsible for it. This approach is complementary to Free Lunch in the sense that HPCTolkit attributes lock contention to threads whereas Free Lunch measure lock related CSP.

`Java.util.concurrent` is a Java API that provides lock-free data structures. JUCProfiler (which is part of MSDK [29]) and JProfiler [22] are able to profile such libraries. Free Lunch does not currently provide this type of profiling. We plan to integrate support for profiling lock-free data structures in future work.

# 7 Conclusion

This paper has presented Free Lunch, a new lock profiler especially designed to identify *phases* of high Critical Section Pressure (CSP) *in-vivo*. Using Free Lunch, we have identified phases of high CSP in Cassandra and in five applications from the DaCapo benchmark suite, the SpecJVM 2008 benchmark suite and the SpecJBB 2005 benchmark. Some of these phases were hidden with previous profilers, which shows that Free Lunch can identify new bottlenecks and reports them back to developer. Thanks to these reports, we were able to improve the performance of the Xalan application by 15% by modifying a single line of code.

We have evaluated Free Lunch on more than thirty applications and shown that it never degrades the performance by more than 6%. This result shows that Free Lunch could be used *in-vivo* to detect phases where a lock hampers the scalability with scenarios that would otherwise not necessarily be tested by a developer *in-vitro*.

# References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open source research community. *IBM System Journal*, 2005.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, 2010.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS*, pages 483–485. ACM, 1967.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI*, 1998.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.

[6] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the linux kernel. In *ALS*, 2000.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[8] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48. ACM, 2013.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2007.

[10] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ISCA*, 2011.

[11] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *IPDPS*, page 6. IEEE, 2003.

[12] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: visualizing scalability bottlenecks in multi-threaded applications. In *OOPSLA*, 2013.

[13] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A substrate for managed runtime environments. In *VEE*, 2010.

[14] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, pages 103–116. ACM, 2009.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ language specification*. Addison-Wesley, 3rd edition, 2005.

[16] IBM Health Center. `http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/`, 2013.

[17] HPROF: A heap/cpu profiling tool. `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`, 2013.

[18] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. HaLock: hardware-assisted lock contention detection in multithreaded applications. In *PACT*, pages 253–262. ACM, 2012.

[19] H. Inoue and T. Nakatani. How a Java VM can get more from a hardware performance monitor. In *OOPSLA*, pages 137–154. ACM, 2009.

[20] Jla homepage. `http://publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp?topic=%2Fcom.ibm.java.doc.igaa%2F_1vg0001143f2181-11a9b04924e-7ff9_1001.html`, 2013.

[21] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[22] JProfiler home page. `http://www.ej-technologies.com/products/jprofiler/overview.html`, 2013.

[23] Java™ Virtual Machine Tool Interface. `http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/`, 2013.

[24] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *OOPSLA*, 2012.

[25] A. Lakshman and P. Malik. Cassandra: Structured storage system on a p2p network. In *PODC*, 2009.

[26] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *ATC*, pages 65–76. USENIX, 2012.

[27] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[28] M. Milenkovic, S. Jones, F. Levine, and E. Pineda. Performance inspector tools with instruction tracing and per-thread / function profiling. In *Linux Symposium*, 2008.

[29] Multicore SDK. `https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=9a29d9f0-11b1-4d29-9359-a6fd9678a2e8`, 2013.

[30] Measuring Lock Contention. `http://0pointer.de/blog/projects/mutrace.html`, 2013.

[31] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In *SoCC*. ACM, 2011.

[32] Safepoints in Hotspot. `http://blog.ragozin.info/2012/10/safepoints-in-hotspot`, 2013.

[33] SPECjbb2005. `http://www.spec.org/jbb2005/`, 2013.

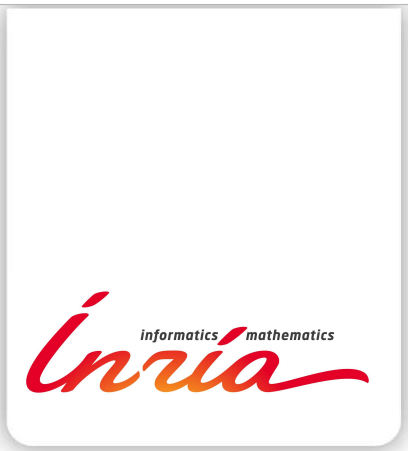
[34] SPECjvm2008. `http://www.spec.org/jvm2008/`, 2013.

[35] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, 2010.

[36] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs. In *OOPSLA*, 2008.

[37] Yourkit home page. `http://www.yourkit.com/`, 2013.

# Contents




**RESEARCH CENTRE**
**PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399